Overview of Java Stream Internals: Construction & Execution

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Understand parallel stream internals, e.g.
- Know what can change & what can’t
- Recognize how a Java stream is constructed & executed
Java Stream Construction & Execution
• Recall that intermediate operations are “lazy”

```
Stream map(Function<…> mapper)
Output f(x)
Stream filter(Predicate<…> pred)
Output g(f(x))
Stream sorted()
Output h(g(f(x)))
R collect(Collector<…> collector)
```

See www.logicbig.com/tutorials/core-java-tutorial/java-util-stream/lazy-evaluation
Recall that intermediate operations are “lazy” 
• i.e., they don’t start to run until a terminal operator is reached

See www.logicbig.com/tutorials/core-java-tutorial/java-util-stream/lazy-evaluation
Java Stream Construction & Execution

- A stream pipeline is constructed at runtime via an internal representation

List<String> ls = ...
List<String> sortedAWords = ls
    .stream()
    .map(String::toUpperCase)
    .filter(s ->
        s.startsWith("A"))
    .sorted()
    .collect(toList());

At runtime a linked list of stream source & intermediate operations is built, one per “stage” in pipeline

A stream pipeline is constructed at runtime via an internal representation.

Each pipeline stage is described by a bitmap of stream flags internally.

<table>
<thead>
<tr>
<th>Stream Flag</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIZED</td>
<td>Size of stream is known</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>Elements of stream are distinct</td>
</tr>
<tr>
<td>SORTED</td>
<td>Elements of the stream are sorted in natural order</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Stream has meaningful encounter order</td>
</tr>
</tbody>
</table>

These flags are a subset of the flags that can be defined by a splitterator.
A stream pipeline is constructed at runtime via an internal representation. Each pipeline stage is described by a bitmap of stream flags internally. Source stage stream flags are derived from spliterator characteristics, e.g.

<table>
<thead>
<tr>
<th>Collection</th>
<th>Sized</th>
<th>Ordered</th>
<th>Sorted</th>
<th>Distinct</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HashSet</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>TreeSet</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Java Stream Construction & Execution

Stream generate() & iterate() methods create streams that are not sized!
Java Stream Construction & Execution

- A stream pipeline is constructed at runtime via an internal representation
  - Each pipeline stage is described by a bitmap of *stream flags* internally
  - Source stage stream flags are derived from spliterator characteristics
  - Each intermediate operation affects the stream flags

<table>
<thead>
<tr>
<th>Input x</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stream map</strong> (Function&lt;...&gt; mapper)</td>
</tr>
<tr>
<td>Output f(x)</td>
</tr>
<tr>
<td><strong>Stream filter</strong> (Predicate&lt;...&gt; pred)</td>
</tr>
<tr>
<td>Output g(f(x))</td>
</tr>
<tr>
<td><strong>Stream sorted()</strong></td>
</tr>
<tr>
<td>Output h(g(f(x)))</td>
</tr>
<tr>
<td>R <strong>collect</strong> (Collector&lt;...&gt; collector)</td>
</tr>
</tbody>
</table>
Java Stream Construction & Execution

- A stream pipeline is constructed at runtime via an internal representation
  - Each pipeline stage is described by a bitmap of stream flags internally
  - Source stage stream flags are derived from spliterator characteristics
  - Each intermediate operation affects the stream flags, e.g.
    - map()
      - Clears SORTED & DISTINCT but keeps SIZED
A stream pipeline is constructed at runtime via an internal representation

- Each pipeline stage is described by a bitmap of *stream flags* internally
- Source stage stream flags are derived from spliterator characteristics
- Each intermediate operation affects the stream flags, e.g.
  - `map()`
  - `filter()`
    - Keeps SORTED & DISTINCT but clears SIZED

```
Java Stream Construction & Execution
```

```
Input x
Stream map(Function<...> mapper)
Output f(x)
Stream filter(Predicate<...> pred)
Output g(f(x))
Stream sorted()
Output h(g(f(x)))
R collect(Collector<...> collector)
```
A stream pipeline is constructed at runtime via an internal representation:

- Each pipeline stage is described by a bitmap of *stream flags* internally.
- Source stage stream flags are derived from spliterator characteristics.
- Each intermediate operation affects the stream flags, e.g.
  - `map()`
  - `filter()`
  - `sorted()`
    - Keeps SIZED & DISTINCT & adds SORTED

```
Input x

Stream map(Function<...> mapper)
  Output f(x)

Stream filter(Predicate<...> pred)
  Output g(f(x))

Stream sorted()
  Output h(g(f(x)))

R collect(Collector<...> collector)
```
A stream pipeline is constructed at runtime via an internal representation:

- Each pipeline stage is described by a bitmap of *stream flags* internally.
- Source stage stream flags are derived from spliterator characteristics.
- Each intermediate operation affects the stream flags.
- The flags at each stage are updated as the pipeline is being constructed.

**Diagram:**

- **Input** $x$:
  - **Stream map** $(\text{Function}\langle\ldots\rangle \, \text{mapper})$:
    - **Output** $f(x)$
  - **Stream filter** $(\text{Predicate}\langle\ldots\rangle \, \text{pred})$:
    - **Output** $g(f(x))$
  - **Stream sorted**:
    - **Output** $h(g(f(x)))$
- **Collect** $(\text{Collector}\langle\ldots\rangle \, \text{collector})$:
  - **Output** $R$
A stream pipeline is constructed at runtime via an internal representation

- Each pipeline stage is described by a bitmap of stream flags internally
- Source stage stream flags are derived from spliterator characteristics
- Each intermediate operation affects the stream flags
- The flags at each stage are updated as the pipeline is being constructed
  - e.g., flags for a previous stage are combined with the current stage’s behavior to derive a new set of flags

Java Stream Construction & Execution
A stream pipeline is constructed at runtime via an internal representation. Each pipeline stage is described by a bitmap of stream flags internally. Source stage stream flags are derived from spliterator characteristics. Each intermediate operation affects the stream flags. The flags at each stage are updated as the pipeline is being constructed. e.g., flags for a previous stage are combined with the current stage’s behavior to derive a new set of flags.

```
Set<String> ts =
    new TreeSet<>(...);

List<String> sortedAWords =
    ts
    .stream()
    .filter(s ->
        s.startsWith("a"))
    .sorted()
    .collect(toList());
```

The streams framework removes redundant operations since the source is already sorted.
Java Stream Construction & Execution

• When terminal operation runs the stream framework picks an execution plan

Input x

Stream map(Function<…> mapper)

Output f(x)

Stream filter(Predicate<…> pred)

Output g(f(x))

Stream sorted()

Output h(g(f(x)))

R collect(Collector<…> collector)

Java Stream Construction & Execution

• When terminal operation runs the stream framework picks an execution plan
• The plan is based on properties of the source & aggregate operations

Input x

Stream map(Function<…> mapper)

Output f(x)

Stream filter(Predicate<…> pred)

Output g(f(x))

Stream sorted()

Output h(g(f(x)))

R collect(Collectors<…> collector)
Java Stream Construction & Execution

- When terminal operation runs the stream framework picks an execution plan
- The plan is based on properties of the source & aggregate operations
- Intermediate operations are divided into two categories

```
Input x
Stream map(Function<…> mapper)
Output f(x)
Stream filter(Predicate<…> pred)
Output g(f(x))
Stream sorted()
Output h(g(f(x)))
R collect(Collector<…> collector)
```
Java Stream Construction & Execution

- When terminal operation runs the stream framework picks an execution plan.
- The plan is based on properties of the source & aggregate operations.
- Intermediate operations are divided into two categories:
  - Stateless
    - e.g., filter(), map(), flatMap(), etc.

A pipeline with only stateless operations runs in one pass (even if it’s parallel).
Java Stream Construction & Execution

- When terminal operation runs the stream framework picks an execution plan
  - The plan is based on properties of the source & aggregate operations
- Intermediate operations are divided into two categories:
  - Stateless
  - Stateful
    - e.g., sorted(), limit(), distinct(), dropWhile(), etc.

A pipeline with stateful operations is divided into sections & runs in multiple passes

$\text{Input } x$

Stream map(Function\text{\(\rightarrow\) mapper})

Output $f(x)$

Stream filter(Predicate\text{\(\rightarrow\) pred})

Output $g(f(x))$

Stream sorted()

Output $h(g(f(x)))$

$R$ collect(Collector\text{\(\rightarrow\) collector})
Java Stream Construction & Execution

- When terminal operation runs the stream framework picks an execution plan
  - The plan is based on properties of the source & aggregate operations
  - Intermediate operations are divided into two categories
  - Terminal operations are also divided into two categories

```
Input x
Stream map(Function<...> mapper)
  Output f(x)
Stream filter(Predicate<...> pred)
  Output g(f(x))
Stream sorted()
  Output h(g(f(x)))
R collect(Collector<...> collector)
```
Java Stream Construction & Execution

- When terminal operation runs the stream framework picks an execution plan:
  - The plan is based on properties of the source & aggregate operations.
  - Intermediate operations are divided into two categories.
  - Terminal operations are also divided into two categories:
    - Non-short-circuiting:
      - e.g., reduce(), collect(), forEach(), etc.

Terminal operation can process data in bulk using spliterator’s forEachRemaining()
Java Stream Construction & Execution

- When terminal operation runs the stream framework picks an execution plan.
- The plan is based on properties of the source & aggregate operations.
- Intermediate operations are divided into two categories.
- Terminal operations are also divided into two categories.
  - Non-short-circuited
  - Short-circuited
    - e.g., anyMatch(), findFirst(), etc.

Terminal operation must process data one element at a time using tryAdvance().
End of Overview of Java Stream Internals: Construction & Execution